# Implementation of a Prototypal Simulator for a Formal Model Based on Pattern Matching and Recombination

Kazuto Tominaga

Tokyo University of Technology, School of Computer Science
1404-1 Katakura, Hachioji, Tokyo 192-0982 Japan
tomi@acm.org

**Abstract.** We implemented a prototypal simulator for a formal model based on pattern matching and recombination, which is a variation of artificial chemistries. The model was designed to capture the properties of complex systems, including those that show self-organizing behavior. The simulator is implemented in Ruby, an object-oriented programming language. The development of the software showed that reducing the number of recombination possibilities by aggregating molecules of the same type contributes to a large performance improvement.

## 1 Introduction

Self-organization phenomena are found in various forms in natural systems. One interesting from of self-organization is organization of structures, such as ontogenesis. For studying such phenomena, artificial chemistries [1] are suitable tools because they model the behavior of systems in terms of reactions among and changes to molecules: structural self-organization can be modelled as processes of forming a large molecule in the course of a sequence of reactions.

We developed a prototypal simulator for an artificial chemistry based on pattern matching and recombination [2]. The model was designed to capture the properties of complex systems, including those that show self-organizing behavior. In this paper, we discuss the implementation of the simulator in detail in order to discern useful techniques for implementing such simulators.

## 2 The Model

The developed software simulates the behavior of systems that are described in terms of a formal model [2], which is a variation of artificial chemistries [1]. In other words, the simulator is an interpreter for "programs" written as definitions in the model. We will explain the model by way of example in this section. A more formal definition is given in [3].

### 2.1 A Simple Example

In this model, a molecule is represented as one or more sequences of elements. Let us suppose we are to have a world that contains series of `ab`, that is, `ab`, `abab`, `ababab`, and so on. Each of these is an *object*, consisting of *elements* `a` and `b`. The object `abab` can be formed by concatenating two objects of the form `ab`. Let us have an object `cd` to help this concatenation, just as a catalyst does in natural chemical reactions.

The world has a multiset of objects that holds the objects existing in the world at each point in time. Let us call this multiset the *working multiset*. This multiset has one `cd` and numerous `ab` at first (Figure 1).

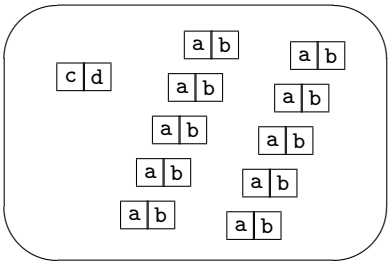



**Fig. 1.** Initial working multiset.

Objects of the form `ab` can be concatenated by the object `cd` by the following process.

1. `cd` attaches at the end of an object that ends with `ab` to form one new compound object.
2. To this object, an object that starts with `ab` attaches and forms a new object.
3. `cd` parts from this object.

Each of the above corresponds to a *recombination rule*. The recombination rules are illustrated in Figure 2. Note that an object can be a single line (such as `cd` in Rule 1) or a stack of lines adjoined top to bottom (such as the compound of $\cdots$`ab` and `cd` in Rule 1).

By applying the recombination rules in an appropriate order, any sequence of the form `abab`$\cdots$ can eventually be obtained. But a problem arises here: the length of the sequence is limited by the number of objects of the form `ab` in the working multiset at the initial state. To obviate this problem, we introduce an infinite supply of `ab`. Let us call it the *source* of `ab`. This source supplies one `ab` at a time to the working multiset. Now it is sufficient for the initial multiset to contain only one object `cd`. This completes the *system* that models our world (Figure 3).

The system is interpreted nondeterministically: any sequence of applications of the rules and supplying `ab` from the source is considered. Then this system models the world in which objects `ab`$\cdots$ of any length are generated.

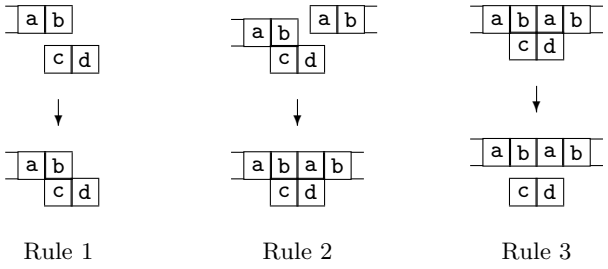



**Fig. 2.** Recombination rules.

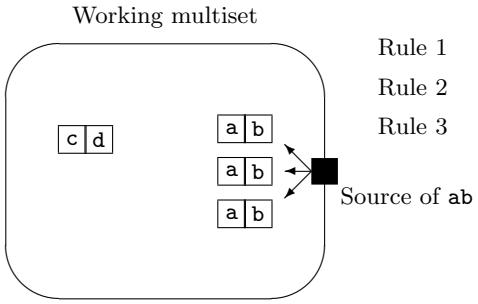
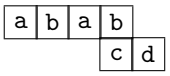



**Fig. 3.** A simple system.

### 2.2 Components of the Model

*Elements.* We have four types of elements, namely, `a`, `b`, `c` and `d`, in the example system.

*Objects.* An *object* is a stack of sequences of elements. An example object is `0:abab/3:cd/`, which is depicted as follows.

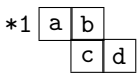


Each sequence of elements, called a *line*, is denoted by "*number*`:`*sequence*`/`". *Number* is the offset of the line from the first line; it is always 0 for the first line.

*Patterns.* A *pattern* matches (or does not match) an object. A pattern is composed of the following components.

- *literal*: a literal is denoted by an element, and it matches the element.
- *element wildcard*: denoted by a digit (`0` to `9`). An element wildcard matches any element.
- *sequence wildcard*: denoted by a digit and an asterisk (`*0` to `*9` and `0*` to `9*`). A sequence wildcard is allowed to appear only at the beginning or at the end of a line. A sequence wildcard that appears at the beginning of a line is denoted by `*0` to `*9`, and one at the end by `0*` to `9*`.

An example pattern is `0:*1ab/1:cd/`, which is depicted as follows. The offset of each line decides the position of the first component that is not a sequence wildcard.




This pattern matches the object `0:abab/3:cd/`, but does not match the object `0:abab/1:cd/`.


matched not matched

*Recombination Rules.* A *recombination rule* is expressed as a pair of two multisets of patterns; let us call these multisets *lhs* and *rhs*. A group of objects matched by *lhs* is transformed to a group of objects that is represented by *rhs*. The element or the sequence of elements matched by a wildcard in *lhs* is placed where the same wildcard appears in *rhs*. To conserve elements in recombination, and to keep the model simple, the following conditions are imposed on recombination rules.

- The multiset of literals in *lhs* equals the multiset of literals in *rhs*.
- A wildcard in *lhs* must be unique in *lhs*.
- A wildcard that appears in *lhs* must appear once and only once in *rhs*.

For example, Rule 2 of the example system


is expressed as the following recombination rule.[1]

$$[\,\texttt{0:*1ab/1:cd/},\ \texttt{0:ab2*/}\,] \rightarrow [\,\texttt{0:*1abab2*/1:cd/}\,]$$

*Sources.* A *source* is defined as an object, and when it operates, it adds one instance of that object to the working multiset.

*Drains.* A *drain* removes objects from the working multiset. A drain is defined as a pattern, and when it operates, it removes one object that is matched by the pattern. The example system has no drains.

[1] We use [ ] to denote a multiset (and { } to denote a set).

### 2.3 Specification of a System

The complete specification of a system is given by defining its elements, sources, drains, initial multiset and recombination rules. The specification of the example system is therefore as follows.

- Elements: { `a`, `b`, `c`, `d` }
- Sources: [ `0:ab/` ]
- Drains: none
- Initial multiset: [ `0:cd/` ]
- Recombination rules:
  [ `0:*1ab/`, `0:cd/` ] → [ `0:*1ab/1:cd/` ] (Rule 1)
  [ `0:*1ab/1:cd/`, `0:ab2*/` ] → [ `0:*1abab2*/1:cd/` ] (Rule 2)
  [ `0:*1abab2*/1:cd/` ] → [ `0:*1abab2*/`, `0:cd/` ] (Rule 3)

### 2.4 Interpretation of the System

A system described in this model is interpreted as follows.

1. Initialize the working multiset to have a specified collection of objects.
2. Do one of the following.
   - Apply one recombination rule to a collection of objects.
   - Operate one source.
   - Operate one drain.
3. Go to Step 2.

The interpretation is nondeterministic: every possible sequence of choices from the initial state is considered. Conditions for termination can be specified (for example, if a specified object appears in the working multiset, or a specified drain operates), or the behavior can be open-ended.

## 3 Implementation

We implemented a prototypal simulator for this model using Ruby [4], an object-oriented programming language with a number of built-in classes — such as character strings, arrays and hashes — that are particularly useful for implementing our system.

To avoid confusion, in the following discussion we will use the word *molecule* to refer to an object in the formal model, and the word *object* to refer to an object in the object-oriented programming language.

The basic design of the implementation is as follows.

- Molecules are kept in an AVL tree [5], an instance of the class `AVL` [6, 7]; this tree represents the working multiset. A key is a string of characters that denotes the molecule, such as `0:abab/1:cd/`, and the stored value is a pair (an array of two elements) $\langle num, mol \rangle$ where $num$ is the number of molecules of that form, and $mol$ is the internal representation of the molecule.
- Operations of the sources, the drains, and the recombination rules are performed in sequence; the decision as to which operates next is made randomly at each step.

### 3.1 Programs

The system is described as a "program" that defines the components of the system. A program for the example system is given below. The program is a code fragment in Ruby, and the components are defined as objects in Ruby.

```
Sources  = ["0:ab/"]
Drains   = []
Initpool = [[1,"0:cd/"]]
Rules    = [
[["0:*1ab/","0:cd/"],["0:*1ab/1:cd/"]],            # Rule 1
[["0:*1ab/1:cd/","0:ab2*/"],["0:*1abab2*/1:cd/"]], # Rule 2
[["0:*1abab2*/1:cd/"],["0:*1abab2*/","0:cd/"]]     # Rule 3
]
SRCPROB = 1
DRNPROB = 1
```

The array `Sources` represents the sources and `Drains` represents the drains. `Initpool` is the initial working multiset, which contains one molecule of the form `0:cd/`. The recombination rules are defined as the array `Rules`: each line, corresponding to one rule, is an array of two elements; the first element is *lhs*, and the second is *rhs*. The above definitions are easily obtained from the specification of the system given in Section 2.3. The constants `SRCPROB` and `DRNPROB` specify probabilities for operating the sources and drains (see the next section).

### 3.2 Dealing with Nondeterminism

In processing programs written in a logic programming language, or in processing terms in a term-rewriting system, the nondeterminism of the system is often dealt with by backtracking. But in simulating a system modelled in an artificial chemistry, backtracking is generally not practical because the number of possible rule applications is often vast due to the large number of molecules contained in the system.

In this implementation, the nondeterminism of the model is dealt with by random choices. In the main loop, the simulator chooses one of the following operations with equal probability.

- To run the sources randomly; each source operates with the probability of 1/`SRCPROB`.
- To run the drains randomly; each drain operates with the probability of 1/`DRNPROB`.
- To apply one of the recombination rules.

If recombination is chosen to be performed, the simulator does the following.

1. For each recombination rule, the simulator lists all the possible combinations of molecules in the multiset to which the recombination rule can be applied.
2. Then the simulator chooses one combination (and its recombination rule) from the aggregate lists, and applies the rule to the molecules.

Because one entry in the AVL tree $\langle num, mol \rangle$ represents all the molecules of the same form, the number of entries in the AVL tree is usually far smaller than the number of molecules. For example, only 41 entries are required to express 1643 molecules in the example execution of solving a 3SAT problem shown in Section 4. This makes the enumerating of possible recombinations less costly.

### 3.3 Pattern Matching

Pattern matching in this model is easier than that of regular expressions for character strings used in general programming languages, since the expressive power of patterns employed in this model is limited, specifically because a sequence wildcard can only appear at either end of a line: it gives fewer possibilities of matching positions.

Instead of adopting the regular-expression class of Ruby, we implemented the pattern-matching function by array operations, aiming future implementations at faster platforms such as those using C [8].

## 4 Examples

An example execution of the program given in Section 3.1 is shown below. The number after `#types` is the number of variations of molecules in the working multiset. `(cmd)` is the prompt for the user; the execution continues with the next iteration when the user pushes the return key.

```
*** iteration 0: #types 1 ***
<1 0:cd/>
(cmd)
run sources
*** iteration 1: #types 2 ***
<1 0:ab/> <1 0:cd/>
(cmd)
run sources
*** iteration 2: #types 2 ***
<2 0:ab/> <1 0:cd/>
(cmd)
recombination
*** iteration 3: #types 2 ***
<1 0:ab/> <1 0:ab/1:cd/>
(cmd)
run drains
*** iteration 4: #types 2 ***
<1 0:ab/> <1 0:ab/1:cd/>
(cmd)
recombination
*** iteration 5: #types 1 ***
<1 0:abab/1:cd/>
(cmd)
recombination
```

```
*** iteration 6: #types 2 ***
<1 0:abab/> <1 0:cd/>
(cmd)
```

After the sixth iteration, a molecule of the form `0:abab/` is generated; the execution can continue further.

Another example is a program to solve a 3SAT problem [2]. This program finds assignments to variables $x_1$, $x_2$, $x_3$ and $x_4$ that satisfy $f = t_1t_2t_3 = (x_1 + x_2 + x_3')(x_1' + x_3 + x_4)(x_2 + x_3 + x_4')$, where $x_i'$ denotes the negation of $x_i$.

```
Sources = [
"0:a/","0:b/","0:c/","0:q/","0:z/",
"0:t/","0:f/","0:d/","0:as/","0:bs/","0:cs/",
"0:aptdddz/","0:apdtddz/","0:apddfdz/",
"0:bpfdddz/","0:bpddtdz/","0:bpdddtz/",
"0:cpdtddz/","0:cpddtdz/","0:cpdddfz/"
]
Drains = []
Initpool = []
Rules = [
# Rules to find assignments that make t1 true
[["0:ap1*/","0:as/","0:a/","0:q/"],["0:ap1*/1:as/0:aq/"]],
[["0:*1t2*/-1:as/0:*3/","0:t/"],["0:*1t2*/0:as/0:*3t/"]],
[["0:*1f2*/-1:as/0:*3/","0:f/"],["0:*1f2*/0:as/0:*3f/"]],
[["0:*1d2*/-1:as/0:*3/","0:d/"],["0:*1d2*/0:as/0:*3d/"]],
[["0:*1z/-1:as/0:*2/","0:z/"],["0:*1z/","0:as/","0:*2z/"]],
[["0:aq1*/","0:bp2*/"],["0:aq1*/0:bp2*/"]],
# Rules to find assignments that make t1t2 true
[["0:aq1*/0:bp2*/","0:bs/","0:b/","0:q/"],["0:aq1*/0:bp2*/1:bs/0:bq/"]],
[["0:*1t2*/0:*3t4*/-1:bs/0:*5/","0:t/"],["0:*1t2*/0:*3t4*/0:bs/0:*5t/"]],
[["0:*1t2*/0:*3d4*/-1:bs/0:*5/","0:t/"],["0:*1t2*/0:*3d4*/0:bs/0:*5t/"]],
[["0:*1d2*/0:*3t4*/-1:bs/0:*5/","0:t/"],["0:*1d2*/0:*3t4*/0:bs/0:*5t/"]],
[["0:*1f2*/0:*3f4*/-1:bs/0:*5/","0:f/"],["0:*1f2*/0:*3f4*/0:bs/0:*5f/"]],
[["0:*1f2*/0:*3d4*/-1:bs/0:*5/","0:f/"],["0:*1f2*/0:*3d4*/0:bs/0:*5f/"]],
[["0:*1d2*/0:*3f4*/-1:bs/0:*5/","0:f/"],["0:*1d2*/0:*3f4*/0:bs/0:*5f/"]],
[["0:*1d2*/0:*3d4*/-1:bs/0:*5/","0:d/"],["0:*1d2*/0:*3d4*/0:bs/0:*5d/"]],
[["0:*1z/0:*2z/-1:bs/0:*3/","0:z/"],["0:*1z/","0:*2z/","0:bs/","0:*3z/"]],
[["0:bq1*/","0:cp2*/"],["0:bq1*/0:cp2*/"]],
# Rules to find assignments that make f = t1t2t3 true
[["0:bq1*/0:cp2*/","0:cs/","0:c/","0:q/"],["0:bq1*/0:cp2*/1:cs/0:cq/"]],
[["0:*1t2*/0:*3t4*/-1:cs/0:*5/","0:t/"],["0:*1t2*/0:*3t4*/0:cs/0:*5t/"]],
[["0:*1t2*/0:*3d4*/-1:cs/0:*5/","0:t/"],["0:*1t2*/0:*3d4*/0:cs/0:*5t/"]],
[["0:*1d2*/0:*3t4*/-1:cs/0:*5/","0:t/"],["0:*1d2*/0:*3t4*/0:cs/0:*5t/"]],
[["0:*1f2*/0:*3f4*/-1:cs/0:*5/","0:f/"],["0:*1f2*/0:*3f4*/0:cs/0:*5f/"]],
[["0:*1f2*/0:*3d4*/-1:cs/0:*5/","0:f/"],["0:*1f2*/0:*3d4*/0:cs/0:*5f/"]],
[["0:*1d2*/0:*3f4*/-1:cs/0:*5/","0:f/"],["0:*1d2*/0:*3f4*/0:cs/0:*5f/"]],
[["0:*1d2*/0:*3d4*/-1:cs/0:*5/","0:d/"],["0:*1d2*/0:*3d4*/0:cs/0:*5d/"]],
[["0:*1z/0:*2z/-1:cs/0:*3/","0:z/"],["0:*1z/","0:*2z/","0:cs/","0:*3z/"]]
]
SRCPROB = 2
DRNPROB = 1
```

The molecules starting with `ap` specified in the initial multiset represent the assignments that satisfy $t_1$. The element `t` means true, `f` false, and `d` don't-care. For example, the molecule `0:aptdddz/` represents the assignment $x_1$ (i.e., true to $x_1$ and don't-care to the other variables), which satisfies $t_1 = (x_1 + x_2 + x_3')$. The molecules starting with `bp` satisfy $t_2$, and those starting with `cp` satisfy $t_3$. The recombination rules gradually make molecules of forms starting with `aq` (satisfying $t_1$)[2], starting with `bq` (satisfying $t_1 t_2$), and starting with `cq` (satisfying $f$, which are solutions to the problem).

An example execution is as follows.

```
*** iteration 0: #types 0 ***
```
*the initial multiset is empty*
```
(cmd)
run sources
*** iteration 1: #types 8 ***
<1 0:apddfdz/> <1 0:apdtddz/> <1 0:b/> <1 0:bpdddtz/> <1 0:bpddtdz/> <1 0
:bs/> <1 0:cs/> <1 0:f/>
(cmd)
run sources
*** iteration 2: #types 15 ***
<1 0:apddfdz/> <2 0:apdtddz/> <1 0:as/> <2 0:b/> <1 0:bpdddtz/> <2 0:bpdd
tdz/> <1 0:bpfdddz/> <1 0:bs/> <1 0:cpdddfz/> <1 0:cpddtdz/> <1 0:cpdtddz
/> <1 0:cs/> <1 0:d/> <2 0:f/> <1 0:t/>
(cmd) cq\w\w\w\wz    ← the user specified a terminating condition
auto run until /cq\w\w\w\wz/ found? [y/N] y
run drains
*** iteration 3: #types 15 ***
<1 0:apddfdz/> <2 0:apdtddz/> <1 0:as/> <2 0:b/> <1 0:bpdddtz/> <2 0:bpdd
tdz/> <1 0:bpfdddz/> <1 0:bs/> <1 0:cpdddfz/> <1 0:cpddtdz/> <1 0:cpdtddz
/> <1 0:cs/> <1 0:d/> <2 0:f/> <1 0:t/>
```
*omitted output of 506 iterations*
```
recombination
*** iteration 509: #types 41 ***
<49 0:a/> <81 0:apddfdz/> <7 0:apddfdz/1:as/0:aq/> <2 0:apddfdz/2:as/0:aq
d/> <1 0:apddfdz/3:as/0:aqdd/> <3 0:apddfdz/4:as/0:aqddf/> <1 0:apddfdz/5
:as/0:aqddfd/> <87 0:apdtddz/> <2 0:apdtddz/1:as/0:aq/> <3 0:apdtddz/2:as
/0:aqd/> <77 0:aptdddz/> <3 0:aptdddz/1:as/0:aq/> <1 0:aptdddz/2:as/0:aqt
/> <1 0:aptdddz/3:as/0:aqtd/> <2 0:aptdddz/5:as/0:aqtddd/> <1 0:aqddfdz/0
:bpddtdz/1:bs/0:bq/> <1 0:aqddfdz/0:bpddtdz/3:bs/0:bqdd/> <1 0:aqddfdz/0:
bpfdddz/> <1 0:aqddfdz/0:bpfdddz/1:bs/0:bq/> <2 0:aqdtddz/0:bpfdddz/4:bs/
0:bqftd/> <1 0:aqtdddz/0:bpddtdz/1:bs/0:bq/> <1 0:aqtdddz/0:bpfdddz/> <68
 0:as/> <79 0:b/> <92 0:bpdddtz/> <93 0:bpddtdz/> <94 0:bpfdddz/> <1 0:bq
ddftz/0:cpdddfz/3:cs/0:cqdd/> <1 0:bqdttdz/> <84 0:bs/> <93 0:c/> <89 0:c
pdddfz/> <110 0:cpddtdz/> <93 0:cpdtddz/> <1 0:cqdttdz/> <97 0:cs/> <29 0
:d/> <94 0:f/> <44 0:q/> <73 0:t/> <80 0:z/>
match for /cq\w\w\w\wz/ found: 0:cqdttdz/
```

[2] This process is redundant because the molecules starting with `ap` represent assignments that satisfy $t_1$, but we introduced this step for symmetry and understandability.

```
clear pattern and continue? [Yn] (n to exit)
```

The molecule `0:cqdttdz/` represents the assignment $x_2x_3$, which makes $f = (x_1 + x_2 + x_3')(x_1' + x_3 + x_4)(x_2 + x_3 + x_4')$ true.

## 5 Discussion

The simulator was implemented as approximately 600 lines of code in Ruby. The main reasons of this easy implementation are as follows.

- A molecule can be represented using a traditional data structure, namely, an array of character strings, and it is processed by traditional string operations, such as pattern matching. The data structure and the operations are supported by the classes and the functions of the programming language.
- A molecule is denoted simply by its structure, and no additional information is necessary, such as its state or location.
- The model does not have the notion of time steps. Because of this, the nondeterminism of the model can be dealt with largely at the discretion of the implementation.

Aggregating molecules of the same form as a pair $\langle num, mol \rangle$ and storing them in an AVL tree contributes to good performance, since it reduces the number of possible combinations that have to be taken into account. In a previous implementation (also in Ruby), the working multiset was represented as an array (one molecule occupied one element of the array), and it solved the same 3SAT problem in approximately 46.5 CPU hours on a Pentium 4 (1.8GHz) PC with 1GB memory; the final working multiset contained 1805 molecules of 114 types. In contrast, this implementation solved the problem in 4.4 CPU seconds on the same machine; the final working multiset had 1643 molecules of 41 types.

The simulator chooses one from among all the possible applications of the recombination rules. Computing all possibilities is more costly than the method in which molecules are first chosen randomly from the working multiset before applicable rules are sought. Furthermore, from the viewpoint of modelling physical or chemical reactions, the latter approach seems more natural. We will develop a simulator adopting this approach.

## 6 Concluding Remarks

We developed prototypal software that simulates the behavior of systems described in a formal model, which is a variation of artificial chemistries. The simulator was implemented easily because the model is based on traditional string operations such as pattern matching, and the platform of the implementation, Ruby, has convenient functions and classes that support those operations.

Although we have not given example descriptions for self-organizing systems, we believe that the model can describe systems that show some kind of structural self-organization. If indeed it can, a simulator of this model will be helpful to study such systems.

## References


1. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries — a review. Artificial Life **7** (2001) 225–275
2. Tominaga, K.: A formal model based on affinity among elements for describing behavior of complex systems. Technical Report UIUCDCS-R-2004-2413, Department of Computer Science, University of Illinois at Urbana-Champaign (2004)
3. Tominaga, K.: A discussion on self-reproducing machines using an artificial chemistry based on pattern matching and recombination (2004) In review.
4. Matsumoto, Y.: Ruby In A Nutshell. O'Reilly & Associates (2001)
5. Adel'son-Vel'skiĭ, G.M., Landis, E.M.: An algorithm for the organization of information. Soviet Mathematics, Doklady **3** (1962) 1259–1263 English translation of Doklady Akademii nauk SSSR, Tom 146, Nos. 1–6, 1962 (Russian).
6. Pfaff, B.: libavl (1999) C library manipulating AVL trees.
7. Pizlo, F.: Ruby/avl (2001) Ruby interface to libavl.
8. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Second edn. Prentice Hall (1988)